1-Les RPC
(Remote Procedure Calls)
2-Le protocole XDR
(eXternal Data Representation)
3-nfs
(network file system)

B.Dupouy

Plan

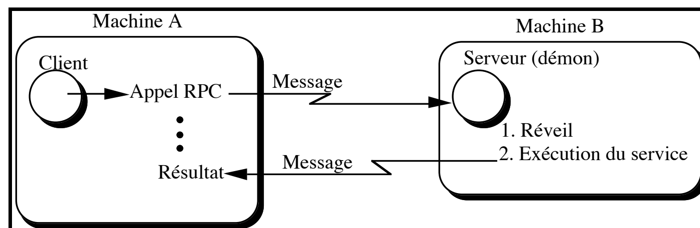
- PRESENTATION DES RPC
  - modèle client/serveur
  - le portmapper
  - la commande rpcinfo

DIFFERENTES COUCHES DU PROTOCOLE RPC
  - La couche haute
  - La couche moyenne
  - La couche basse

- DIVERS
  - Diffusion
  - Authentification
  - RPCGEN

L'appel de procédure à distance

• Principe : mécanisme client/serveur.



• Les paramétres sont :

– envoyés dans un message : *marshalling*

– regroupés dans un objet unique (*call by copy-restore*)

– normalisés au format XDR.

• En cas de :

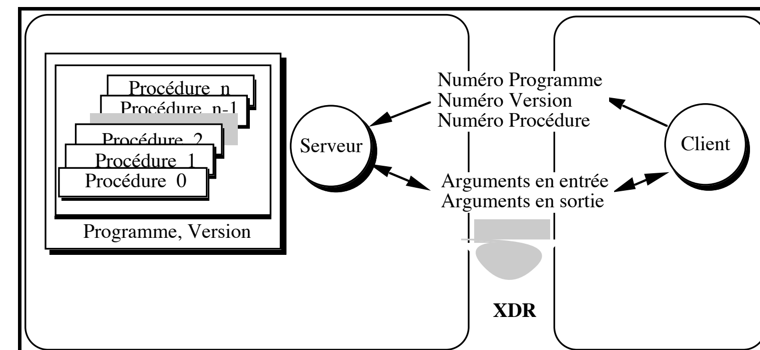
– perte de son message d'acquittement, le serveur garantit l'idempotence de l'exécution

– défaillance du serveur on garantit l'exécution *at least once*

Le modèle client/serveur des RPC

• Le serveur est repéré par le nom du programme qu'il propose (un entier sur 32 bits).

• Le serveur propose des procédures numérotées et regroupées dans un programme :



• La procédure 0 doit toujours exister :

– c'est une procédure vide

– elle sert à vérifier l'existence du service

• Exclusion mutuelle entre les procédures d'un même programme :

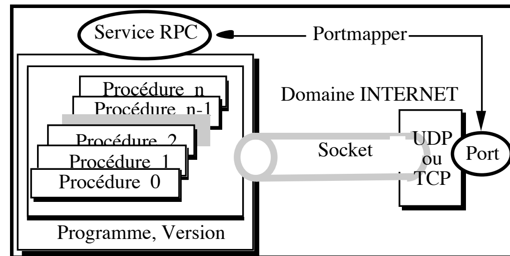
– une seule procédure peut s'exécuter à la fois

Rôle du portmapper

• Comment trouver le numéro de port d'un service dont on ne connaît que le nom ?

• Le **`portmapper`** **(serveur de noms)** renvoie le numéro de port du service (le programme) dont on lui a communiqué le nom

• Le **`portmapper`** a pour numéro de port 111



• Les serveurs RPC sont lancés par inetd, le portmapper doit être démarré avant

• Liste des services enregistrés : `/usr/etc/rpcinfo -p`

La commande `rpcinfo`

`/usr/etc/rpcinfo`

`rpcinfo -p Machine`	Donne les services disponibles sur le site `Machine`
`rpcinfo -b Prog Vers`	Donne la liste des sites fournissant le service `Prog`
`rpcinfo -u Machine Prog Vers`	Appel à la procédure 0 du programme `Prog` sur `Machine`

Les services RPC

`rpcinfo -p`

```
program vers proto   port
 100000    2   tcp    111  portmapper
 100000    2   udp    111  portmapper
 100007    2   tcp    660  ypbind
 100007    2   udp    662  ypbind
 100029    1   udp    660  keyserv
 100003    2   udp   2049  nfs
 100005    2   udp    707  mountd
 100005    2   tcp    710  mountd
 ...
 100021    3   tcp    724  nlockmgr
 100021    3   udp   1031  nlockmgr
 100020    2   udp   1032  llockmgr
 100020    2   tcp    729  llockmgr
 100021    2   tcp    732  nlockmgr
 100021    2   udp   1033  nlockmgr
 100011    1   udp   1034  rquotad
 100001    3   udp   1036  rstatd
 100001    4   udp   1036  rstatd
 100002    1   udp   1037  rusersd
 100002    2   udp   1037  rusersd
 100012    1   udp   1038  sprayd
 100008    1   udp   1039  walld
 100068    2   udp   1040  cmsd
 100068    3   udp   1040  cmsd
 100083    1   tcp   1028
 150001    1   udp         pcnfsd
```


Les fonctions de la couche haute

`getmaster`	donne le nom du maître de NIS
`getrpcport`	donne le numéro du port associé à un serveur
`havedisk`	indique si une machine à un disque
`rstats`	donne des indices de performance sur le système visé
`rusers (shell)`	donne des informations sur les utilisateurs d'une machine
`rwall (shell)`	pour écrire sur la machine cible
`yppasswd (shell)`	met à jour le mot de passe dans la base de données NIS

Remarque

- on charge cette bibliothèque avec l'option `-lrpcsvc`

La bibliothèque RPC

• Deux familles de fonctions : la couche moyenne et la couche basse

• Couche moyenne :

- pas de choix de protocole (c'est toujours UDP)
- pas de choix d'authentification, de temporisation ...

Couche moyenne	
Côté client	Côté serveur
`callrpc ()`	`registerrpc()` `svc_run ()`

Couche basse	
Côté client	Côté serveur

`clntxxx_create ()`
`clnt_destroy ()`
`clnt_call ()`
`clnt_control ()`
`clnt_freeres ()`
	`svcxxx_create ()`
	`svc_destroy ()`
	`svc_getargs ()`
	`svc_freeargs ()`
	`svc_register ()`
	`svc_sendreply ()`
	`svc_getrequest ()`

La couche moyenne

Côté client	**callrpc** (Site, Num_Prog, Num_Vers, Num_Proc, filtreXDR_out, Pteur vers variable out, filtreXDR_in, Pteur vers variable in)
Côté serveur	**registerrpc** (Num_Prog, Num_Vers, Num_Proc, Nom fonction locale, filtreXDR_out, filtreXDR_in) **svc_run**()

• Remarques :

– on ne passe qu'**un** argument en entrée et en sortie, il faudra donc les regrouper en **structures** s'ils sont nombreux

– utiliser `rpcgen` pour générer les filtres XDR des structures

• Exemple de serveur :

```
...
struct Etat_Machine {
     int Nb_Proc;
     int Nb_Util;
     int Disque;
     } Etat;
...
/* Enregistrement du service, quand le service est
appele, la procedure    Etat est automatiquement lancee.
*/
registerrpc (STAT_PROG, STAT_VERS, STAT_PROC,
             Etat, xdr_void, xdr_Etat_Machine);
svc_run ();
```

La couche basse (1)

• Permet de gérer :

- le mode de transport : UDP ou TCP
- des appels non-bloquants
- les valeurs de time-out et la fréquences des relances
- diffusion ,batching , call back

• Communication avec les sockets (ou TLI) :

- les données sont transmises via des flots XDR :
    - encode · read ou sendto,
    - decode ·write ou recvfrom
- deux structures de données font l'interface avec le transport :

Pour le serveur	Pour le client
`SVCXPRT` (définie dans `rpc/svc.h`)	`CLIENT` (définie dans `rpc/clnt.h`)

La couche basse (2)

• Ce qu'il faut faire :

Côté serveur	Côté client
1-Créér un pointeur vers un objet du type `SVCXPRT`	1-Créér un pointeur vers un objet du type `CLIENT`
2-Enregistrer le service	2-Appeler la procédure distante
3-Se mettre en attente des appels distants	3-Détruire le pointeur une fois les résultats obtenus
Et bien sûr, avoir préparé les procédures...	

• L' objet du type `SVCXPRT` contient, entre autres :

– le numéro de socket (si pas créé, indiquer `RPC_ANYSOCK`)

– le numéro de port

Schéma de programmation du serveur

1-initialisation du pointeur sur une structure `SVCXPRT`	`SVCXPRT *Ptr_SVC`
	`...`
	`Ptr_SVC``=svcudp_create(Socket)`
	ou
	`Ptr_SVC``=svctcp_create(Socket,···)`

destruction des références précédentes	`pmap_unset(···)`

2-enregistrement **en une seule fois** de toutes les procédures du service.	`/* La fonction aiguillage Services contient toutes les procédures offertes par le servuer */`
	
	`svc_register (`**`Ptr_SVC`**`,Prog,Vers,Services,Prot)`

3-mise en attente du serveur	`svc_run()`

écrire la fonction aiguillage : `Services`	`Services (`**`Ptr_Req,Ptr_SVC`**`)`
	`struct svc_req *Ptr_Req;`
	`SVCXPRT *Ptr_SVC;`

- Fonction `Services :`

`svc_req` pointée par `Ptr_Req` contient les paramétres envoyés par l'appel RPC distant : numéro de programme, version, numéro de procédure et authentification

La couche basse (3)

• Correspondance entre fonctions de la couche moyenne et de la couche basse :

Couche moyenne	Couche basse

`callrpc`	`clntudp_create,clnttcp_create`
	suivi de :
	`clnt_call`

`registerrpc`	`svcudp_create, svctcp_create`
	suivi de :
	`svc_register`

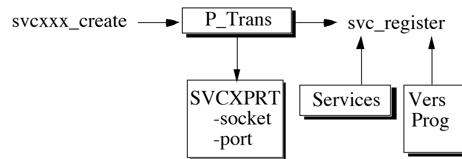
`svc_run`	`svc_run`

• Un client écrit avec la couche moyenne peut appeler un serveur écrit avec la couche basse et vice-versa

Structures de données utilisées

• Par `svc_register` :

```
P_Trans=svctcp_create(RPC_ANYSOCK,0,0);
...
/*Enregistrer le numero de port par appel au
portmapper*/
if (! svc_register(P_Trans , PROG ,VERS ,Services,
PROTO))
```

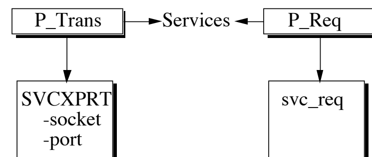


• Par `Services`, la fonction aiguillage :

- un pointeur sur un objet du type `svc_req` où elle trouve la demande faite par le client (numéro de procédure demandée),

- un pointeur sur un objet du type `SVCXPRT` pour échanger des paramètres avec le client, via les fonctions :

  - `svc_getargs` (lit les entrées)
  - `svc_sendreply` (range les résultats).



Canevas de `Services`

• Syntaxe

```
Services (Ptr_Req, Ptr_Svc)
      struct svc_req    *Ptr_Req;
      SVCXPRT    *Ptr_Svc;
```

• Partie aiguillage :

```
switch (Ptr_Req -> rq_proc)
      {
      case service_1 :
      /* Decoder paramètres in du format XDR au  local*/
      svc_getargs (Ptr_Svc, xdr_entree, entree)

      /* Appeler la fonction locale gérant ce service */
      service_1 (...)

      /* Encoder les résultats au format XDR*/
      svc_sendreply (Ptr_Svc, xdr_result, result)
      break;
      }

      ...

      case service_i :
      {
      ...
      }

      ...

      default : svcerr_noproc (Ptr_Svc);
```

Serveur : Transfert des paramètres

• `svc_getargs` extrait (via le filtre `xdr_entree`) les paramétres du flot XDR et les range à l'adresse `entree`

```
svc_getargs (Ptr_Svc, xdr_entree, entree)
```

• `svc_sendreply` encode (via le filtre `xdr_result`) les paramétres dans le flot XDR qui les transmet sur le port précisé par `Ptr_Svc->xp_port`

```
svc_sendreply(Ptr_Svc, xdr_result,result)
```

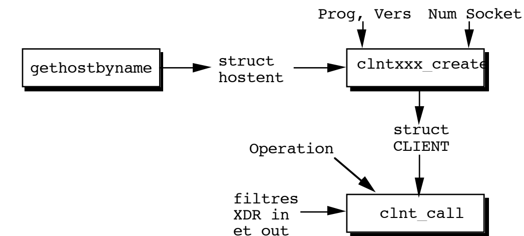

Programmation du client

obtention de l'adresse du site à appeler	`gethostbyname`

Initialiser le pointeur sur la structure `CLIENT`	`CLIENT *Ptr_Cli`
	`Ptr_Cli = clntudp_create (···)`
	ou
	`Ptr_Cli = clnttcp_create (···)`

Appel d'une procédure distante	`clnt_call (Ptr_Cli , ···)`
fin de connexion	`clnt_destroy(Ptr_Cli )`

• Structures de données mises en œuvre :



Programmation du client

• Création d'un pointeur vers un objet du type `CLIENT` :

```
CLIENT   *Ptr_Cl
...
Ptr_Cl = clnttcp_create (&Serveur,
         NUM_PROG,NUM_VERS,&socket,0,0)
```

• Appel de la procédure distante :

```
Etat = clnt_call(Ptr_Cl,NUM_PROC,xdr_ent,
                 tab_ent,xdr_res,&res,
                 timeout);
    ...
clnt_destroy(Cl);
```


Diffusion d'un appel

• Appel en diffusion au serveur précédent :

```
clnt_broadcast (PROG, VERS, PROC, xdr_void,
     xdr_Etat_Machine, &Etat_Serv, Traite_Res);
```

• `Traite_Res` est appelée à chaque réponse reçue, si renvoie `TRUE`, alors arrêt de la diffusion.

• Syntaxe de `Traite_Res`

```
/*Ptr_Etat arrive ici décodé par XDR
  Ptr_Hote décrit le site qui a répondu
*/
bool_t Traite_Res (Ptr_Etat, Ptr_Hote)
     struct Etat_Machine *Ptr_Etat;
     struct sockaddr_in *Ptr_Hote;
```

• Structure de cette fonction

```
{
struct in_addr      *Num;
struct hostent      *Exped;
...
Num = &(*Ptr_Hote).sin_addr;
Exped = gethostbyaddr (Num, sizeof (*Num), AF_INET);

Meil_Res.Nb_Proc = Ptr_Etat->Nb_Proc ;
Meil_Res.Nb_Util = Ptr_Etat->Nb_Util;
Meil_Res.Disque  = Ptr_Etat->Disque;
...
```

• Attention : cet appel ne franchit PAS les passerelles.

Authentification Unix

- Chaque appel RPC transporte avec lui une structure d'authentification.
- Ce que fait le serveur dans `Services` pour contrôler les accès :

```
void Services(rqstp,Transp)
            struct svc_req    *rqstp;
            SVCXPRT           *Transp;
{
struct authunix_parms *unix_cred;
switch (rqstp->rq_cred.oa_flavor)
/* choix du type d'auth */
     {
     case AUTH_UNIX :
     unix_cred = rqstp->rq_clntcred;
     uid = unix_cred->aup_uid;
     gid = unix_cred->aup_gid;
/*acces interdit aux clients dont uid diff. de 2200*/
     if (uid != 2200)
            {
            fprintf(stderr,"Acces interdit ");
            return;
            }
```

- Ce doit faire le client pour s'identifier :

```
Cl->cl_auth = authunix_create_default();
...
clnt_call (Cl,Operation, xdr_ent, tab_ent, xdr_result,
            &result, timeout);
...
```


Rpcgen

- Précompilateur de langage C qui engendre quatre (ou trois) fichiers, à partir d'une description donnée dans un fichier `fichier.x` :
  - un fichier à inclure dans les codes du serveur et des clients (`fichier.h`),
  - un squelette du code du serveur (`fichier_svc.c`),
  - les procédures réalisant les appels distants pour les clients (`fichier_clnt.c`).
  - si nécéssaire, un fichier de définition des filtres XDR (`fichier_xdr.c`),

- Il reste à écrire deux fichiers :
  - un fichier contenant le code des procédures fournies par le serveur qui complétera `fichier_svc.c`, lui-même contenant déja la fonction `main`.
  - le code du client qui se servira de `fichier_clnt.c`, ici on écrit le `main`.

Rpcgen

• Inconvénients : manque de souplesse

• Avantages :

- prise en charge de toutes les interfaces de transport
- définitions de filtres

• A utiliser SYSTEMATIQUEMENT pour définir les filtres même si on gère soi-même toute l'application

rpcgen : définition de filtre

• Fichier `fitre.x` :

```
struct st1 {
      int val1;
      float val2 [10];
      string val3<10>;
      float val4[5];
};
```

Fichier `fitre.h`

```
#include
<rpc/types.h>


struct st1 {
    int val1;
    float val2[10];
    char *val3;
    float val4[5];
};


typedef  struct  st1
st1;
bool_t xdr_st1();
```

Fichier `fitre_xdr.c`

```
#include <rpc/rpc.h>
#include "filtre.h"

bool_t
xdr_st1(xdrs, objp)
    XDR *xdrs;
    st1 *objp;
{
if (!xdr_int(xdrs, &objp->val1)) {
        return (FALSE);
}
if  (!xdr_vector(xdrs,  (char  *)objp->val2,
10, sizeof(float), xdr_float)) {
        return (FALSE);
    }
if (!xdr_string(xdrs, &objp->val3, 10)) {
        return (FALSE);
    }
if  (!xdr_vector(xdrs,  (char  *)objp->val4,
5, sizeof(float), xdr_float)) {
        return (FALSE);
    }
    return (TRUE);
```

rpcgen : *exemple complet*

• **Exemple** fichier `arith.x :`

```
/*
Deux services sont proposes : carre et racine
*/
program BID_PROG {
        version BID_VERS  {
                int BID_CARRE (int) = 1;
                int BID_RAC (int) = 2;
        } = 1;
} = 22;
```

• La commande `rpcgen arith.x` engendre :

- `arith.h`
- `arith_svc.c`
- `arith_clnt.c`
- PAS de `arith_xdr.c (types standards)`

• Il restera donc à écrire les fichiers :

- `arith.c` qui compléte `arith_svc.c`
- `cli_arith.c` qui compléte `arith_clnt.c`

Fichiers créés par `rpcgen` :

• Le fichier `arith.h`

C'est la liste des *include* dont auront besoin le client et le serveur.

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include <rpc/types.h>

#define BID_PROG ((u_long)22)
#define BID_VERS ((u_long)1)

#define BID_CARRE ((u_long)1)
extern int *bid_carre_1();

#define BID_RAC ((u_long)2)
extern int *bid_rac_1();
```

Fichiers créés par `rpcgen`: `main` du serveur

- Le fichier `arith_svc.c` (Squelette du code du serveur) contient :
  - la création du `socket`,
  - l'enregistrement du service, la mise en attente,
  - et surtout la fonction d'aiguillage (`bid_prog_1`).

- Il restera à écrire le code des fonctions qui s'appeleront :
  - `bid_carre_1`
  - `bid_rac_1`.

```
/*Please do not edit this file generated using rpcgen */
...
main()
{
SVCXPRT     *transp;
pmap_unset (BID_PROG, BID_VERS);
transp = svcudp_create (RPC_ANYSOCK);
svc_register (transp, BID_PROG, BID_VERS, bid_prog_1,
              IPPROTO_UDP)
...
```


Fichiers créés par rpcgen (fonction aiguillage du serveur)

```
static void
bid_prog_1(rqstp, transp)
        struct svc_req *rqstp;
        register SVCXPRT *transp;
{
        union {
                int bid_carre_1_arg;
                int bid_rac_1_arg;
        } argument;
        char *result;
        bool_t (*xdr_argument)(), (*xdr_result)();
        char *(*local)();

        switch (rqstp->rq_proc) {
        case NULLPROC:
               ...

        case BID_CARRE:
                xdr_argument = xdr_int;
                xdr_result   = xdr_int;
                local = (char *(*)()) bid_carre_1;
                break;

        case BID_RAC:
                xdr_argument = xdr_int;
                xdr_result   = xdr_int;
                local = (char *(*)()) bid_rac_1;
                break;

        default:
                svcerr_noproc(transp);
                return;
        }
              ...
              svc_getargs(transp, xdr_argument, &argument))
              ...
              result = (*local)(&argument, rqstp);
              ...
              svc_freeargs(transp, xdr_argument, &argument))
              ...
```

Fichiers créés par rpcgen (appels RPC du client)

• Le fichier `arith_clnt.c`

Il s'agit des appels RPC (`clnt_call`) du client vers le serveur.

```
#include <rpc/rpc.h>
#include "arith.h"
...
/****************************************************/
int * bid_carre_1(argp, clnt)
      int *argp;
      CLIENT *clnt;
{
      static int res;
      bzero((char *)&res, sizeof(res));
      if(clnt_call (clnt,BID_CARRE,xdr_int,argp,
             xdr_int,&res,TIMEOUT) != RPC_SUCCESS)
             {
             return (NULL);
             }
      return (&res);
}
/****************************************************/
int * bid_rac_1(argp, clnt)
      int *argp;
      CLIENT *clnt;
{
      static int res;
      bzero((char *)&res, sizeof(res));
      if(clnt_call (clnt,BID_RAC,xdr_int,argp,
               xdr_int,&res,TIMEOUT) != RPC_SUCCESS)
      ...
      return (&res);
}
```


Fonctions écrites par l'utilisateur (serveur)

• Le fichier `arith.c`

• Ce sont les fonctions spécifiques du service à ajouter dans le serveur.

• **bid_rac_1**

```
int  *bid_rac_1 (Ptr_Ent)
int  *Ptr_Ent;
{
       static int i;
       double sqrt();

       i= (int) sqrt ( (double) (*Ptr_Ent) );
       return ( &i );
}
```

• **bid_carre_1**

```
int  *bid_carre_1(Ptr_Ent)
int  *Ptr_Ent;
{
       static int i;
       i=   (*Ptr_Ent) * (*Ptr_Ent) ;
       return ( &i );
}
```

`main` écrit par l'utilisateur (client)

• Le fichier `cli_arith.c` qui contient la fonction **main** du client, les appels RPC sont fournis par `RPCGEN`.

```
main (argc, argv)
int argc;
char *argv[];
      {
      CLIENT *cl;
      ...
      Param = atoi (argv[2]);
      ...
      cl=clnt_create(Hote,BID_PROG,BID_VERS, "udp");
      ...
      printf (" Que faire (rac ou carre)?\n");
      scanf ("%c",&c);
      switch (c)
      {
      case 'c' :
      Resultat = bid_carre_1 (&Param, cl);
      ...
      printf("carre%d:%d",Param,*Resultat);
      exit (0);
      case 'r' :
      Resultat = bid_rac_1 (&Param, cl);
      ...
```


Le protocole XDR

B.Dupouy

Plan

- POSITION DU PROBLEME
- BIBLIOTHEQUE XDR
  - Filtres
  - Flots de données
  - Support physique
- EXEMPLE
- RPCGEN

XDR : position du problème

lire.c	ecrire.c

`#include <stdio.h>`
`main ()`
`{`
`int i;`
`fread`
`(&i,sizeof(i),1,stdin);`
`printf ("%d\n",i);`
`}`
	`#include <stdio.h>`
	`main ()`
	`{`
	`int i=24;`
	`fwrite`
	`(&i,sizeof(i),1,stdout);`
	`}`

La commande :

```
rsh Machine ecrire | lire
```

donnera un résultat différent de 24 si l'architecture de `Machine` est différente de celle de la machine locale.

Exemples :

big endian / little endian

format des flottants (IEEE ou autres)

Bibliothèque XDR

Les fonctions y sont de deux types :

• filtres de conversion et transferts de données

• création et gestion des flots XDR

Flot XDR :

• suite d'octets dans laquelle les données sont rangées au format XDR

Filtre XDR :

• convertit (encode ou **sérialise**) du format local à XDR

• convertit (décode ou **désérialise**) de XDR au format local

• se construisent pour tout type d'objets à partir des filtres de base

• rôle de `rpcgen`

Les flots XDR (XDR streams)

Rendent les filtres indépendants du medium de transfert :

• un filtre s'adresse uniquement à un flot

• le flot fait l'interface avec le support physique (transparence)

Implantation :

• le flot est implanté sous forme d'un `XDR handle`

• ce handle contient les opérations à appliquer au flot :

```
XDR_ENCODE, XDR_DECODE, XDR_FREE
```

Du flot XDR au support physique

Le flot XDR peut être associé à:

• de la mémoire :

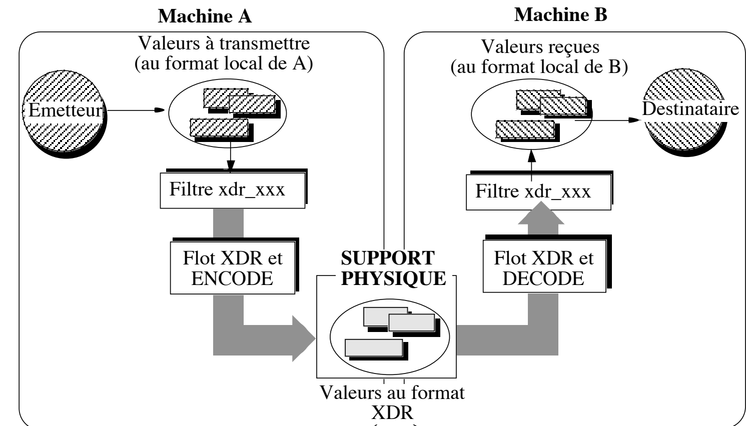
```
xdrmem_create (XDR_Ptr, Tab, Dim, Oper)
    XDR_Ptr      *xdr_handle
    char       *Tab
    u_int   Taille
    enum xdr_op  Oper
```

• un fichier :

```
xdrstdio_create (XDR_Ptr, Desc_Fic, Oper)
    XDR_Ptr      *xdr_handle
    *FILE   Desc_Fic
    enum xdr_op  Oper
```

• ce fichier peut être un **socket**

Schéma de fonctionnement



Filtres XDR

Format général des filtres

```
xdr_xxx   (XDR_Ptr, Ptr_obj)
    XDR      *XDR_Ptr
    xxx      *Ptr_obj
```

où `xxx` peut être :

char, int, float,
bytes
vector
array
reference
...

Quand on applique un filtre sur un flot XDR :

flot défini `DECODE`	flot défini `ENCODE`
données **lues** dans le flot	données **écrites** dans le flot

Exemple : flot et filtre sur fichiers

Deux processus échangent un entier et un flottant :

Code de l'écrivain	Code du lecteur

`XDR xdrs;`
`long val1=10;`
`float val2=4.456789;`
	`...`

`/* ouverture en écriture*/` `fp=fopen(FIC,"w");`	`/* ouverture en lecture */` `fp=fopen(FIC,"r");`

`/* creation flot XDR*/`
`xdrstdio_create`
`(&xdrs,fp,XDR_ENCODE);`
	`/* creation flot XDR*/`
	`xdrstdio_create`
	`(&xdrs,fp,XDR_DECODE);`

`/* ecriture d'un entier*/` `xdr_long(&xdrs,&val1);`	`/* lecture d'un entier*/` `xdr_long(&xdrs,&val1);`
`/* ecriture d'un flottant*/` `xdr_float(&xdrs,&val2);`	`/* lecture d'un flottant*/` `xdr_float(&xdrs,&val2);`

	`/* ecrire resultat obtenu*/`
	`printf`
	`("On lit %d %f\n",val1,val2);`

Construction de filtres

Exemple de filtre pour une structure `Couple` composée d'un entier et d'un réel.

La structure de données :

```
struct Couple
{
int Entier,
float Reel
};
```

La fonction créant le nouveau filtre de données :

```
bool_t xdr_Couple  (XDR_Ptr, Couple_Ptr)
    XDR*            XDR_Ptr;
    struct Couple*  Couple_Ptr;

{
if ( xdr_int   (XDR_Ptr, &Couple_Ptr->Entier) &&
     xdr_float (XDR_Ptr, &Couple_Ptr->Reel) ) return
TRUE;
return FALSE;
}
```


Création de filtres avec `rpcgen`

Fichier `bidon.x` donné par l'utilisateur :

```
struct st1
      {
      int val1;
      float val2 [10];
      string val3 <10>;
      float val4[5];
      };
```

Fichier `bidon.h` généré par `rpcgen` :

```
struct st1
      {
      int val1;
      float val2[10];
      char *val3;
      float val4[5];
      };

typedef struct st1 st1;

bool_t xdr_st1();
```

Fichier `bidon_xdr.c` produit par `rpcgen` :

```
#include "bidon.h"
bool_t xdr_st1 (xdrs, objp)
XDR *xdrs;
st1 *objp;

{
if (!xdr_int(xdrs, &objp->val1)) return (FALSE);
if
(!xdr_vector(xdrs, (char *)objp->val2, 10,
sizeof(float), xdr_float)) return (FALSE);
if (!xdr_string(xdrs, &objp->val3, 10)) return (FALSE);
if
(!xdr_vector(xdrs, (char *)objp->val4, 5, sizeof(float),
xdr_float)) return (FALSE);

return (TRUE);
}
```

Network File System NFS

B.Dupouy

NFS :
Objectifs

• Transparence d'accès à des fichiers distants,

• Implantation simple par extension du SGF local : ajout de démons (comme les IPC de Berkeley), et non pas par la modification du noyau.

• Robustesse (!) : pas d'état pour le protocole mount.

• Rapidité : caches disques à la fois sur le serveur et le client.

• Facilité d'administration.

NFS et l'OSI :

Numéro de la couche	Nom de la couche	Protocole
7	Application	NFS
6	Présentation	XDR
5	Session	RPC
4	Transport	TCP/UDP

VFS	indépendance vis à vis des différents SGF
RPC	indépendance vis à vis du système d'exploitation
XDR	indépendance vis à vis de la machine

FONCTONNEMENT CLIENT/SERVEUR

• Un site peut être à la fois serveur **et** client.

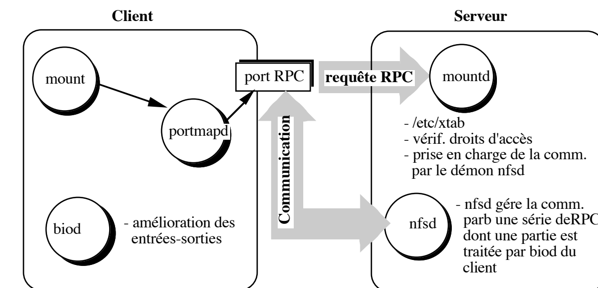
• Les serveurs NFS sont sans états (*stateless*) :

- la panne d'un client n'affecte pas le comportement du serveur,

- celle d'un serveur n'astreint pas ses clients à un traitement spécial.

- défaut :un client ne distingue pas un site en panne d'un site ayant temps de réponse particulièrement long, et il va réitérer ses demandes pendant un délai *(time-out*) souvent insupportable pour l'utilisateur !

Schéma de principe



• Le client a l'initiative du *binding* vers le serveur avec la commande `mount` qui va lire `/etc/fstab.`

• Le serveur restreint les accès distants en gérant le fichier `/etc/exports` qui est lu par `exportfs.`

IMPLANTATION SOUS UNIX

• Ajout de deux modules:

- l'interface VFS,

- la structure vnode.

• vnode

L'exécution d'une commande passe par cette structure vnode :

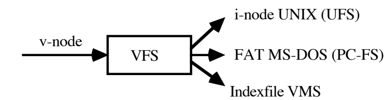
• si le fichier appartient à un *file-system* Unix, alors le vnode fera référence au *inode* correspondant,

• sinon le vnode permattra de référencer le fichier DOS, VMS, ...

VFS

• Définit les opérations pouvant être réalisées sur un fichier. Sous Unix, la plupart des opérations classiques sont autorisées (pas de `od` sur un répertoire distant)

• Ces opérations sont réalisées par des *remote procedure calls*



• Procure une interface orientée objet au SGF, c'est à dire qu'il définit des structures et des fonctions d'accès à ces structures.

• Un SGF quelconque qui utilise le couple vnode/VFS peut être "accroché" au SGF. On ajoute le SGF distant comme on ajoute un périphérique grâce à un nouveau driver.

Les RPC impliquées

• Chaque requête NFS (service 100003) est implantée sous forme d'appel à une ou plusieurs fonction(s) RPC,

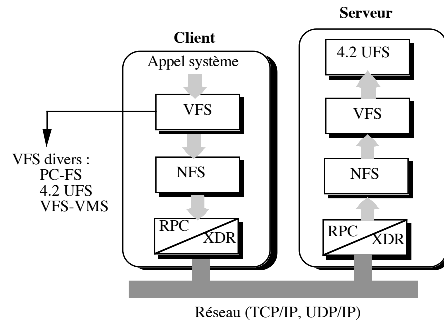
• Exemple : une copie (`cp`) engendre cinq requête aux procédures du service NFS :

```
- getattr
- lookup
- read
- create
- write.
```


Liste des RPC de NFS :

Nom de la procédure	Rôle de la procedure
`null`	ne retourne rien, pour mesurer le temps d'un aller-retour vers un serveur
`lookup (dirfh,nom)`	retourne (fh, attr), c'est à dire un nouveau pointeur de fichier et les attributs de ce fichier
`create (dirfh,nom,attr)`	retourne (nouv_fh, attr), crée un nouveau fichier et renvoie son filehandle et ses attributs.
`remove (dirfh,nom)`	retourne (etat), retire un fichier d'un répertoire
`getattr (fh)`	retourne (attr), c'est à dire les attributs du fichier pointé par fh
`read (fh depla,nbre)`	renvoie (attr, donnees) et lit nbre octets à partir de l'octet depla
`write (fh,depla,nbre,donnees)`	renvoie (attr) et ecrit nbre octets à partir de l'octet depla.
`link (dirfh,nom,tofh,tonom)`	renvoie (etat)
`symlink (dirfh,nom,chaine)`	renvoie (etat)
`readlink (fh)`	renvoie (chaine)
`mkdir (dirfh,nom,attr)`	renvoie (fh, nouv_attr)
`rmdir (dirfh,nom)`	renvoie (etat)
`readdir (dirfh,cookie,nbre)`	renvoie (entrees)
`statfs (fh)`	renvoie (etat)

Schéma de fonctionnement :



• Démons sur le serveur :

- `nfsd`, lance sur le serveur tous les démons gérant les clients (paramétre `nservers`)

- `mountd` est créé par `rc.` au chargement du système.

• Démon sur le client :

- `biod` gère les caches disque locaux parallèlement à ceux gérés sur le serveur (*read ahead* et *write behind*)

Extrait de la commande `ps -axl` exécutée sur la machine `eole` :

```
      F UID   PID  PPID CP PRI NI  SZ  RSS WCHAN    STAT TT  TIME COMMAND
...
   88001   0    90     1  0   1  0  16    0 nfs_dnlc I    ?   1:30  (biod)
   88001   0    91     1  0   1  0  16    0 nfs_dnlc I    ?   1:34  (biod)
   88001   0    92     1  0   1  0  16    0 nfs_dnlc I    ?   1:31  (biod)
   88001   0    93     1  0   1  0  16    0 nfs_dnlc I    ?   1:33  (biod)
...
  488001   0   119     1  0   1  0  28    0 socket   S    ?   3:23  (nfsd)
   88000   0   120     1  0   1  0  64    0 select   IW   ?   0:30 rpc.mountd
   88001   0   122   119  0   1  0  28    0 socket   S    ?   3:15  (nfsd)
   88001   0   123   119  0   1  0  28    0 socket   S    ?   3:04  (nfsd)
   88001   0   124   119  0   1  0  28    0 socket   S    ?   3:21  (nfsd)
   88001   0   125   119  0   1  0  28    0 socket   S    ?   3:09  (nfsd)
   88001   0   126   119  0   1  0  28    0 socket   S    ?   3:35  (nfsd)
   88001   0   127   119  0   1  0  28    0 socket   S    ?   3:30  (nfsd)
   88001   0   128   119  0   1  0  28    0 socket   S    ?   3:14  (nfsd)
```

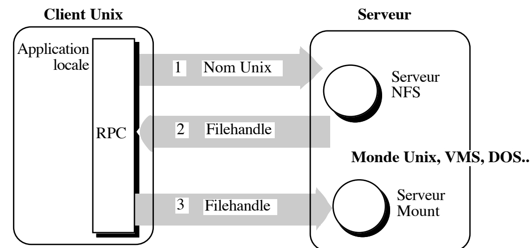
Quand le client ouvre un fichier sans liens sur le serveur, alors :• Le client créé un `nfs.xxx` qui sera détruit au close,• Si le client s'arrête avant le close `nfs.xxx` RESTE sur le serveur !

nfs.xxx est un pointeur vers le fichier.

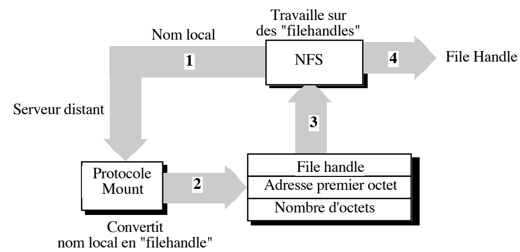
Schéma de fonctionnement :

Echanges entre ces processus

Rôle de mount :



Utilisation des filehandle par NFS :



II.7 Fichiers impliqués

commande	fichier utilisé
`exportfs`	exports (entrée), xtab (sortie)
`mount`	fstab (entrée), mtab (sortie)
`getexport...`	xtab

SUIVI DE NFS

• Principales commandes

Commande	Effet
`/usr/etc/nfsstat`	donne toutes les informations possibles sur nfs, sans rien initialiser
`/usr/etc/netstat -c`	donne le nombre de retransmissions, doit être ≤ 0,5%.
`rpcinfo -p Serveur`	donne la liste des services RPC sur Serveur
`rpcinfo -u Serveur mount`	pour savoir si mountd tourne sur Serveur
`readmont`	sur un serveur, donne la liste des volumes montés par des clients
`showmount -e`	sur un serveur, donne la liste des fichiers exportés

SUIVI DE NFS
(NFSSTAT)

• Sortie de la commande `/usr/etc/nfsstat`

```
Server rpc:
calls      badcalls   nullrecv   badlen     xdrcall
596404     0          0          0          0

Client rpc:
calls      badcalls  retrans    badxid     timeout    wait       newcred
timers
186847     59         99          15          123        0          0
33254


Server  nfs:
calls       badcalls
596404      449
null        getattr     setattr     root        lookup      readlink    read
991  0%     303682 50%  3453  0%    0  0%       136728 22%  41894  7%   66078 11%
wrcache     write       create      remove      rename      link        symlink
0  0%       13915  2%   2334  0%    1467  0%    462  0%     252  0%     0  0%
mkdir       rmdir       readdir     fsstat
471  0%     10  0%      23973  4%   694  0%

Client  nfs:
calls       badcalls    nclget      nclsleep
186837      54          186825      0
null        getattr     setattr     root        lookup      readlink    read
0  0%       22215 11%   1230  0%    0  0%       36269 19%   70  0%      50832 27%
wrcache     write       create      remove      rename      link        symlink
0  0%       67753 36%   2148  1%    1529  0%    292  0%     0  0%       0  0%
mkdir       rmdir       readdir     fsstat
51  0%      51  0%      4350  2%    47  0%
```


SUIVI DE NFS
(RPCINFO)

• Utilisation de la commande `/usr/etc/rpcinfo`

```
usr/etc/rpcinfo -p

   program   vers proto   port
    100000    2   tcp     111  portmapper
    100000    2   udp     111  portmapper
    ...
    100003    2   udp    2049  nfs
    100005    1   udp     722  mountd
    100005    2   udp     722  mountd
    100005    1   tcp     725  mountd
    100005    2   tcp     725  mountd
    ...
    150001    1   udp     751  pcnfsd
    ...
    100012    1   udp    1041  sprayd
    ...
```

On notera le démon `psnfsd`, pour gérer l'interface nfs avec MS-DOS.

SUIVI DE NFS
(RPCINFO -B)

• /usr/etc/rpcinfo -b 100005 1

```
137.194.48.48 orion
137.194.48.26 rigel
137.194.8.2 liszt
137.194.8.5 chopin
137.194.40.5 aruna
137.194.24.1 dali
137.194.24.10 warhol
...
137.194.48.30 mambo
137.194.48.31 cocody
137.194.32.1 inf
137.194.40.6 uranie
...
137.194.48.27 nuwanda
137.194.48.48 orion
137.194.48.26 rigel
137.194.32.3 enst
137.194.8.2 liszt
```

• Parmi ces machines fournissant le service 100005 (NFS), on trouve 4 sites VAX/VMS : `orion, liszt, rigel` et `chopin`.

EXEMPLE DE SUIVI DE NFS (1)

• df sur pegase

```
Filesystem              kbytes    used    avail  capacity  Mounted on

/dev/sd0a                 9391    4572     3879    54%    /
/dev/sd0g                57999   48179     4020    92%    /usr
/dev/sd1a               191022   50521   130949    28%    /home/pegase
ulysse:/usr/share       132123  124648        0   105%    /usr/share
ulysse:/usr/local       164143  138273     9455    94%    /usr/local
tape_ulysse              18633    7805    10828    42%    /rfs/tape_ulysse
tape_helios              12052    8536     3516    71%    /rfs/tape_helios
ulysse:/var/spool/mail
                        102938   81879    10765    88%    /auto/var/spool/mail
zeus:/cal/zeus         1217519  414686   790658    34%    /auto/cal/zeus
ulysse:/usr/bin/X11     164143  145549     2179    99%    /auto/usr/bin/X11
ulysse:/usr/lbin        238728  157241    76712    67%    /auto/usr/lbin
stheno:/inf/stheno      191022  164805    16665    91%    /auto/inf/stheno
tango:/sig/tango1       246099  196418    25072    89%    /auto/sig/tango1
zeus:/usr/lib/X11       238559  183057    31647    85%    /auto/usr/lib/X11
```

• /etc/fstab sur pegase

```
/dev/sd0a               /                       4.2 rw,nosuid 1 1
/dev/sd0g               /usr                    4.2 rw            1 2
/dev/sd1a               /home/pegase            4.2 rw            1 3
#
ulysse:/usr/share       /usr/share              nfs ro,bg,soft0 0
ulysse:/usr/local       /usr/local              nfs rw,bg,soft0 0
#ulysse:/usr/lbin       /usr/lbin           nfs rw,bg,soft0 0
#ulysse:/usr/bin/X11    /usr/bin/X11            nfs ro,bg,soft0 0
#ulysse:/usr/lib/X11    /usr/lib/X11            nfs ro,bg,soft0 0
#
# RFS
#
tape_ulysse         /rfs/tape_ulysse        rfs rw,bg,soft  0 0
tape_helios         /rfs/tape_helios        rfs rw,bg,soft  0 0
#tape_matrix        /rfs/tape_matrix        rfs rw,bg,soft  0 0
```

EXEMPLE DE SUIVI DE NFS (2)

• démons nfs sur pegase

```
  88001    0     66      1   0    1   0   16     0 nfs_dnlc I     ?    0:04  (biod)
  88001    0     67      1   0    1   0   16     0 nfs_dnlc S     ?    0:04  (biod)
  88001    0     68      1   0    1   0   16     0 nfs_dnlc I     ?    0:04  (biod)
  88001    0     69      1   0    1   0   16     0 nfs_dnlc I     ?    0:04  (biod)

 488001    0     93      1   0    1   0   28     0 socket   S     ?    0:13  (nfsd)
  88001    0     94     93   0    1   0   28     0 socket   S     ?    0:13  (nfsd)
  88001    0     95     93   0    1   0   28     0 socket   S     ?    0:15  (nfsd)
  88001    0     96     93   0    1   0   28     0 socket   S     ?    0:14  (nfsd)
  88001    0     97     93   0    1   0   28     0 socket   S     ?    0:15  (nfsd)
  88001    0     98     93   0    1   0   28     0 socket   S     ?    0:16  (nfsd)
  88001    0     99     93   0    1   0   28     0 socket   S     ?    0:15  (nfsd)
  88001    0    101     93   0    1   0   28     0 socket   S     ?    0:16  (nfsd)
```


EXEMPLE DE SUIVI DE NFS (3)

• /etc/exportfs sur ulysse

```
/           -root=morgane,access=morgane
/usr        -root=morgane:enst,access=ENST
/usr/share  -root=morgane:enst:jupiter,access=ENST:Eurecom
/usr/local  -root=morgane:enst,access=ENST:Eurecom
/usr/lbin   -root=morgane:enst,access=ENST
/usr/public -root=morgane,access=ENST:Eurecom
/var/spool  -root=morgane:enst:jupiter,access=ENST
/var/games  -root=morgane:erebe,access=ENST
#
# partitions inf
#
/home/ulysse-root=morgane:enst:jupiter,access=Inf:Cal:Res
/home/heracles  -root=morgane:enst:jupiter,access=Inf:Cal
/home/castor-root=morgane:enst,access=Inf:Cal
/home/pollux-root=morgane:enst,access=Inf:Cal
/home/jason     -root=morgane:enst,access=Inf:Cal
/home/argos     -root=morgane:enst,access=Inf:Cal
/home/hades     -root=morgane:enst,access=Inf:Cal
/home/public-
    root=morgane:enst:jupiter,access=Inf:Cal:Sig:Res:Eurecom
/home/archive   -root=morgane:enst:jupiter,access=ENST:Eurecom
/home/apple     -root=morgane:enst:jupiter,access=ENST
/home/cdrom     -ro,root=morgane,access=ENST
#
#
# Stations sans disque
#
#/export/root/morgane           -access=morgane,root=morgane
#/export/swap/morgane           -access=morgane,root=morgane
#/export/exec/sun4.sunos.4.1 -access=morgane
```